Object-Oriented Programming

Lesson 2

Brief review of data types
Passing arguments to a function
Dynamic object creation
Default constructors
Destructors
Pointer arithmetic

Data types

Data types define the way you use the storage (memory) in the programs. By specifying a data type, you tell the sompiler how to create a particular piece of storage, and how to manipulate that storage.

Data types can be built-in or abstract.

A built-in data type is one that the compiler intrinsically understands, one that is wired directly into the compiler.

The compiler "learns" how to handle abstract data types by reading header files containing class definitions.

Basic built-in types – char, int, float, double, bool.

Specifiers – modify the meaning of the basic built-in types, i.e. modify the maximum and minimum values that a data type will hold.

long, short, signed, unsigned.

Pointers

Whenever you run a program, it is first loaded into the computer's memory. Thus, all elements of your program are located somewhere in memory. Each program element can be uniquely distinguished from all other elements by its address.

Operator & tells the address of an element.

Pointers

```
#include <iostream>
using namespace std;

void f(int )
{}

int main()
{
  cout << "f(): " << &f << endl;

  int ar[3];
  cout << "ar[0]:    " << &ar[0] << endl;
  cout << "ar[1]:    " << &ar[1] << endl;
  cout << "ar[2]:    " << &ar[2] << endl;
  cout << sizeof(ar) << endl;
}
```

```
f(): 003B1311
ar[0]:    001CFB30
ar[1]:    001CFB34
ar[2]:    001CFB38
12
```

Pointers

Variable that holds the address is a pointer.

When you define a pointer, you must specify the type of variable it points to.

```
int* pInt;
CBook* pBook;
```

Example:

What is declared here?

```
int* ipa, ipb, ipc;
```

```
int* ipa;
int* ipb;
int* ipc;
```

```
int* ipa,* ipb, * ipc;
```

Pointers

To access a variable through a pointer, you dereference the pointer using the same operator that you used to define it.

```
int a = 47;
int* ipa = &a;

*ipa = 74;
```

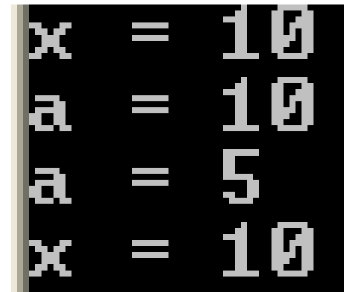
Passing arguments to a function

Pass-by-value: when you pass an argument to a function, a copy of that argument is made inside the function.

```
#include <iostream>
using namespace std;

void f(int a)
{
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}

int main()
{ int x = 10;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
}
```



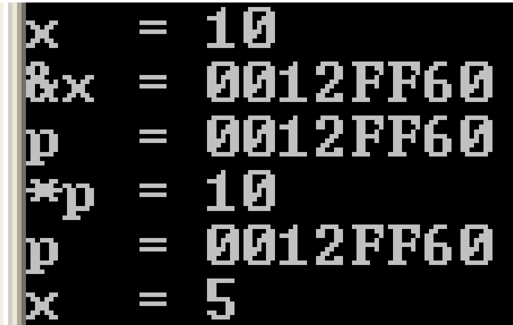
a is a local variable that only exists for the duration of the function call.

When you're inside **f( )**, **x** is the *outside object*, and changing the local variable does not affect the outside object (since they are two separate locations in storage).

It is not possible to pass arrays by value!

Passing arguments to a function

Pass-by-pointer: you pass a *pointer* into a function instead of an ordinary value, actually passing an alias to the outside object, enabling the function to modify that outside object.




```
#include <iostream>
using namespace std;

void f(int* p)
{
  cout << "p  = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 5;
  cout << "p  = " << p << endl;
}

int main()
{
  int x = 10;
  cout << "x  = " << x << endl;
  cout << "&x = " << &x << endl;
  f(&x);
  cout << "x  = " << x << endl;
}
```

Passing arguments to a function

Pass-by-reference: you pass a reference into a function (the function gets a reference to the object, i.e. its address).

The difference between references and pointers is that *calling* a function that takes references is cleaner, syntactically, than calling a function that takes pointers.

```
void f(int& r)
{ cout << "r  = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r  = " << r << endl;
}

int main()
{ int x = 10;
  cout << "x  = " << x << endl;
  cout << "&x = " << &x << endl;
  f(x);
  cout << "x  = " << x << endl;
}
```

```
x  = 10
&x = 0012FF60
r  = 10
&r = 0012FF60
r  = 5
x  = 5
```

Inside **f( )**, if you just say '**r**' (which would produce the address if **r** were a pointer) you get *the value in the variable that* ***r*** *references*. If you assign to **r**, you actually assign to the variable that **r** references.

Object creation

When a C++ object is created, two events occur:

1. Storage is allocated for the object.
2. The constructor is called to initialize that storage.

Storage can be allocated in different ways:

- before the program begins, in the static storage area. This storage exists for the life of the program.

- on the stack whenever a particular execution point is reached (an opening brace). That storage is released automatically at the complementary execution point (the closing brace).

- from a pool of memory called the heap (or free store). This is called dynamic memory allocation and is used when you do not know exactly how many objects you need when you are writing the program. You can decide at any time that you want some memory and how much you need. You are also responsible for determining when to release the memory, which means the lifetime of that memory can be as long as you choose – it isn't determined by scope.

Dynamic storage allocation

To allocate memory dynamically at runtime, C provides functions in its standard library: malloc( ), calloc( ), realloc( ), free( ).

The solution in C++ is to combine all the actions necessary to create and destroy an object into two operators called new and delete.

Operator new

Operator new allocates enough storage on the heap to hold the object and calls the constructor for that storage.

The default new checks to make sure the memory allocation was successful before passing the address to the constructor, so you don't have to explicitly determine if the call was successful. The new returns the address of the created and properly initialized object.

```
int* ip = new int;
CBook* pBook = new CBook ("C++", 2014);
```

Operator new

In C++, when you create arrays of objects on the stack or on the heap, the constructor is called for each object in the array. There's one constraint, however: there must be a default constructor, because a constructor with no arguments must be called for every object.

```
CBook* books = new CBook[5];
```

Operator delete

The complement to the new-expression is the delete-expression, which first calls the destructor and then releases the memory (often with a call to **free( )**). Just as a new-expression returns a pointer to the object, a delete-expression requires the address of an object.

If you forget to delete an object created on the heap, you get a memory leak.

```
int* ip = new int;                          ⇔   delete ip;
CBook* pBook = new CBook ("C++", 2014); ⇔       delete pBook;
```

If you delete a void pointer, the only thing that happens is the memory gets released, because there's no type information and no way for the compiler to know what destructor to call.

Operator delete

When you delete an array of objects, you have to give the compiler the information that the pointer is actually the starting address of an array:

```
CBook* books = new CBook[5];
delete [] books;
```

The empty brackets tell the compiler to generate code that fetches the number of objects in the array, stored somewhere when the array is created, and calls the destructor for that many array objects.

If you forget to use empty brackets, all the used by the array memory will be released, but only the first object desctructor will be called.

If you try to delete a null pointer, nothing happens.

```
int main()
{
   CBook* pBook = new CBook ("C++", 2014);
   delete pBook;
   pBook = 0;
   delete pBook;
}
```

Default constructor

A default constructor is one that can be called with no arguments.

The default constructor is so important that *if* (and only if) there are no constructors in a class, the compiler will automatically create one for you.

```
class V
{
        int i;
        // public: V (int ii) { i = ii; }
};
int main()
{
        V v, v2[10];
}
```

If any constructors are defined, however, and there's no default constructor, the code above will generate compile-time errors.

Default constructor

The default constructors generated by the compiler do not perform* initialization of the class members. If you want the memory to be initialized to zero, you must do it yourself by writing the default constructor explicitly.

Although the compiler will create a default constructor for you, the behavior of the compiler-synthesized constructor is rarely what you want. In general, you should define your constructors explicitly and not allow the compiler to do it for you.

Constructors e destructors

If a class does not have a destructor, the compiler will generate one.

The destructor synthesized by the compiler does not free* memory occupied by the class members.

In general, you should not leave this task to the compiler and should define explicitly the destructor for every class.

It is not possible to obtain the address of a constructor.

It is not possible to obtain the address of a destructor.

Pointer arithmetic

Pointer arithmetic refers to the application of some of the arithmetic operators to pointers.

```
#include <iostream>
#include "Book.h"

using namespace std;
int main()
{
    int i[10];
    int* ip = i;
    cout << "ip = " << ip << endl;
    ip++;
    cout << "ip = " << ip << endl;

    CBook b[10];
    CBook* bp = b;
    cout << "bp = " << bp << endl;
    bp++;
    cout << "bp = " << bp << endl;
}
```

```
ip = 0012FF30
ip = 0012FF34
bp = 0012FECC
bp = 0012FED4
```

4 bytes

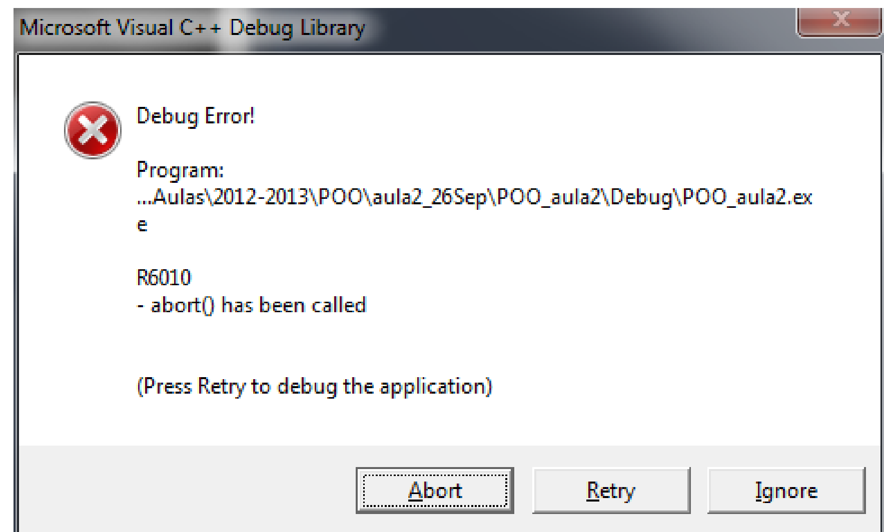
8 bytes

The operator ++, instead of adding 1, makes the pointer to point to the next value (i.e. moves the pointer *x* bytes where *x* is the size of each array element).

Debugging

In the standard header file <cassert> you'll find assert(), which is a convenient debugging macro. When you use assert, you give it an argument that is an expression you are "asserting to be true."
The preprocessor generates code that will test the assertion. If the assertion isn't true, the program will stop after issuing an error message telling you what the assertion was and that it failed.

```
#include <cassert>
int main()
{
        bool test = false;
        assert(test);
}
```



The macro can only be used in the Debug configuration! In the Release configuration the macro will not work!

Bibliography

Bruce Eckel, Thinking in C++, 2nd edition, MindView, Inc., 2003

=> chapters 3, 6, 13